BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data

Paul G. Brown Peter J. Haas

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120-6099
{pbrown1,phaas}@us.ibm.com

Abstract

We present the BHUNT scheme for automatically discovering algebraic constraints between pairs of columns in relational data. The constraints may be "fuzzy" in that they hold for most, but not all, of the records, and the columns may be in the same table or different tables. Such constraints are of interest in the context of both data mining and query optimization, and the BHUNT methodology can potentially be adapted to discover fuzzy functional dependencies and other useful relationships. BHUNT first identifies candidate sets of column value pairs that are likely to satisfy an algebraic constraint. This discovery process exploits both system catalog information and data samples, and employs pruning heuristics to control processing costs. For each candidate, BHUNT constructs algebraic constraints by applying statistical histogramming, segmentation, or clustering techniques to samples of column values. Using results from the theory of tolerance intervals, the sample sizes can be chosen to control the number of "exception" records that fail to satisfy the discovered constraints. In query-optimization mode, BHUNT can automatically partition the data into normal and exception records. During subsequent query processing, queries can be modified to incorporate the constraints; the optimizer uses the constraints to identify new, more efficient access paths. The results are then combined with the results of executing the original query against the (small) set of exception records. Experiments on a very large database using a prototype implementation of BHUNT show reductions in table accesses of up to two orders of magnitude, leading to speedups in query processing by up to a factor of 6.8.

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

1 Introduction and Overview

Commercial DBMS vendors increasingly view autonomic and self-managing technologies as crucial for maintaining the usability and decreasing the ownership costs of their systems [10, 11]. Self-tuning database systems have also been receiving renewed attention from the research community; see, e.g., [22] and references therein. Query optimizers that actively learn about relationships in the data are an important component of this emerging technology.

In this paper we provide a new data-driven technique called BHUNT (for Bump HUNTer) that automatically discovers algebraic relationships between attributes and provides this information to the optimizer in the form of constraint predicates, along with an estimate of the predicates' selectivity. The optimizer can use this information in the usual way to improve cost estimates. Perhaps more importantly, knowledge of the discovered predicates can also provide new access plans for the optimizer's consideration. As we show empirically, the new access paths can lead to substantial speedups in query processing. Such predicates also allow the database administrator (DBA) to consider alternative physical organizations of the data, such as the creation of materialized views and/or indexes, or the use of alternative partitioning strategies. Finally, the predicates may be of interest in their own right, providing new insights into application data. BHUNT can potentially be extended to discover other relationships such as fuzzy functional dependencies.

1.1 Two Examples

To make the discussion concrete, we give two examples of algebraic constraint predicates and their use in speeding up query processing.

Example 1 Consider a hypothetical sales database that contains tables `orders` and `deliveries` as in Figure 1, and suppose that the database contains many year's worth of data. A casual inspection of the columns `orders.shipDate` and `deliveries.deliveryDate` may not reveal any

meaningful relationships, but if we execute the SQL query

```
SELECT DAYS(deliveries.deliveryDate)
   - DAYS(orders.shipDate)
FROM orders, deliveries
WHERE orders.orderID = deliveries.orderID
```

and plot a histogram of the resulting data points, we could well obtain a plot as in Figure 2. It is apparent from the figure that, except for a small number of outlier points, the data satisfy the predicate

```
    (deliveryDate BETWEEN shipDate + 2 DAYS
    AND shipDate + 5 DAYS)
OR (deliveryDate BETWEEN shipDate + 12 DAYS
    AND shipDate + 19 DAYS)
OR (deliveryDate BETWEEN shipDate + 31 DAYS
    AND shipDate + 35 DAYS)
```
(1)

The three clauses in the predicate — equivalently, the three "bumps" in the histogram — correspond to three shipping methods. Knowledge of this predicate can help the optimizer choose an efficient method for joining the `orders` and `deliveries` tables. E.g., consider the query

```
SELECT COUNT(*)
FROM orders, deliveries
WHERE orders.shipDate BETWEEN '2003-07-01'
   AND '2003-07-05'
AND deliveries.deliveryTime > '17:00'
AND orders.orderID = deliveries.orderID
```

Suppose that there exist indexes on columns `orders.orderID`, `deliveries.orderID`, and `deliveries.deliveryDate` but not on `orders.shipDate`. Combining the predicate in the foregoing query with the predicate in (1), we obtain the following new local predicate for the `deliveries` table:

```
(deliveryDate BETWEEN '2003-07-01' + 2 DAYS
   AND '2003-07-05' + 5 DAYS)
OR
(deliveryDate BETWEEN '2003-07-01' + 12 DAYS
   AND '2003-07-05' + 19 DAYS)
OR
(deliveryDate BETWEEN '2003-07-01' + 31 DAYS
   AND '2003-07-05' + 35 DAYS)
```
(2)

One possible access plan first uses the index on `deliveries.deliveryDate` to efficiently apply both the predicate in (2) and the predicate on `deliveries.deliveryTime` to the `deliveries` table. Then, for each qualifying row, the plan uses the index on `orders.orderID` to find the matching record in the `orders` table and apply the original predicate on `orders.shipDate`. Observe that this access plan is not available to the optimizer without knowledge of the predicate in (1). Because the number of qualifying rows from the `deliveries` table is small, this access plan should be relatively efficient. It is clearly more efficient than the plan that first applies the predicate on `orders.shipDate` and then joins each qualifying row with its matching row from the `deliveries` table. □

Example 2 Consider the example of the Section 1, but now suppose that the `deliveryDate` column is located in the `orders` table, as shown in Figure 3. Also suppose

orderID	shipDate
2A5	2001-01-03
3C2	2001-04-15
3B8	2002-11-25
2E1	2002-10-31
3D6	2002-07-25
...	...

orders

orderID	deliveryDate	deliveryTime
2A5	2001-01-06	09:50
3C2	2001-04-27	13:00
3B8	2002-12-10	11:20
2E1	2002-12-02	16:10
3D6	2002-07-29	08:50
...	...	...

deliveries

Figure 1: Two tables in a sales database

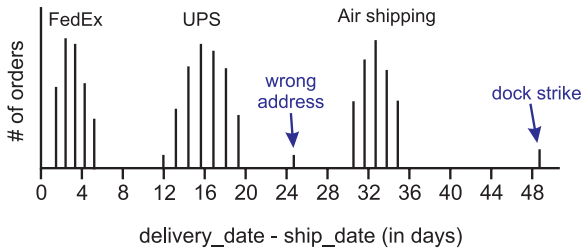

Figure 2: Histogram of shipping delays

that the `orders` table is horizontally range-partitioned on `deliveryDate` across a number of parallel processing nodes. Finally, suppose that we wish to process the following query:

```
SELECT COUNT(*)
FROM orders
WHERE orders.shipDate = '2003-07-01'
```

If we can discover the predicate in (1), then we can derive the predicate

```
deliveryDate BETWEEN '2003-07-01' + 2 DAYS
   AND '2003-07-01' + 5 DAYS
OR
deliveryDate BETWEEN '2003-07-01' + 12 DAYS
   AND '2003-07-01' + 19 DAYS
OR
deliveryDate BETWEEN '2003-07-01' + 31 DAYS
   AND '2003-07-01' + 35 DAYS.
```
(3)

The optimizer can then exploit this information to speed up processing by identifying those partitions that potentially contain rows satisfying the predicate in (3), and hence satisfying the original query. Processing can then be restricted to the identified partitions. □

1.2 Algebraic Constraints

The predicate in (1) asserts an algebraic relationship between a pair of columns. In general, an algebraic relationship on numerical attributes a_1 and a_2 has the mathematical form

$$a_1 \oplus a_2 \in I, \tag{4}$$

orderID	shipDate	deliveryDate
2A5	2001-01-03	2001-01-06
3C2	2001-04-15	2001-04-27
3B8	2002-11-25	2002-12-10
2E1	2002-10-31	2002-12-02
3D6	2002-07-25	2002-07-29
...	...	...

orders

Figure 3: Alternate version of `orders` table

where $\oplus$ is an algebraic operator, i.e., $+$, $-$, $\times$, or $/$, and I is a subset of the real numbers. To completely specify the relationship, we need to specify which particular a_1 values get paired with which particular a_2 values to form the set of number pairs acted on by the $\oplus$ operator. We do this by specifying a *pairing rule* P. In the simplest case, the columns lie in the same table R and each a_1 value is paired with the a_2 value in the same row. The pairing rule is then trivial, and we denote it by the symbol $\emptyset_R$. When the columns lie in tables R and S, then P is simply a two-table join predicate that is satisfied for each pair of tuples (r, s) such that $(r.a_1, s.a_2)$ is one of the number pairs acted on by $\oplus$. We allow tables R and S to coincide, so that P can be a self-join predicate. In general, there can be more than one pairing rule between two specified columns, and multiple pairs (a_1, a_2) can share the same pairing rule. An example of the former situation occurs when two columns are in the same table and specific column values are paired if they occur in the same row or are paired if they appear in different rows that are related via a self-join predicate. An example of the latter situation occurs when P represents a join between tables R and S, and an algebraic relationship exists both between $R.a_1$ and $S.a_2$ and between $R.b_1$ and $S.b_2$.

In light of the foregoing discussion, we specify an *algebraic constraint* as a 5-tuple

$$AC = (a_1, a_2, P, \oplus, I),$$

where a_1, a_2, $\oplus$, and I are as in (4) and P is a pairing rule. For example, the algebraic constraint in Example 1 is specified by taking a_1 as `deliveries.deliveryDate`, a_2 as `orders.shipDate`, $\oplus$ as the subtraction operator, P as the predicate

$$\texttt{orders.orderID} = \texttt{deliveries.orderID},$$

and

$$I = \{\, 2, 3, 4, 5 \,\} \cup \{\, 12, 13, \ldots, 19 \,\} \cup \{\, 31, 32, 33, 34, 35 \,\}.$$

The algebraic constraint in Example 2 is specified almost identically, except that now we take a_1 as `orders.deliveryDate` and P as the trivial pairing rule $\emptyset_{\texttt{orders}}$.

We restrict attention to the case in which $I = I_1 \cup \cdots \cup I_k$ for some $k \geq 1$, where the sets in the union are mutually disjoint and either each I_j is an interval of the real line or each I_j is an interval of the integers. Thus we focus on algebraic constraints that correspond to disjunctive range predicates. In this case we often write the algebraic constraint as

$$AC = (a_1, a_2, P, \oplus, I_1, \ldots, I_k).$$

Useful algebraic constraints abound in real-world data sets, but are often hidden from the DBMS for one of the following reasons:

- The constraint is inherent to the problem domain but unknown to both the application developer and the DBA.
- The constraint is enforced by the application that uses the DBMS and not by the DBMS itself.
- The DBA knows about the constraint but chooses not to enforce it for reasons of cost.
- The constraint is *fuzzy* in that most, but not all, of the data satisfy the constraint. The constraint is therefore not a standard DBMS "rule" per se. (This is the case in Examples 1 and 2.)

1.3 Overview of BHUNT

The BHUNT scheme automatically and efficiently finds and exploits hidden, fuzzy algebraic constraints. BHUNT proceeds by executing the following steps:

1. Find *candidates* of the form $C = (a_1, a_2, P, \oplus)$. This process involves, among other things, finding declared or undeclared key columns and then finding columns related to the key columns via an inclusion dependency.
2. For each candidate, construct the algebraic constraint (i.e., construct the intervals $I_1, I_2, \ldots, I_k$) by applying statistical histogramming, segmentation, or clustering techniques to a sample of the column values. The sample size is selected to control the number of "exception" records that fail to satisfy the constraint.
3. Identify the most useful set of constraints, and create "exception tables" to hold all of the exception records.
4. During query processing, modify the queries to incorporate the constraints — the optimizer uses the constraints to identify new, more efficient access paths. Then combine the results with the results of executing the original query against the (small) exception table.

Steps 1 and 2 are always executed in BHUNT. Steps 3 and 4 are executed whenever BHUNT is used for query optimization. In this latter setting, Steps 1–3 are executed prior to query processing in much the same way as statistics-collection utilities are invoked in order to populate the system catalog. Step 4 is executed either when a query is compiled or run. BHUNT is flexible in that it does not require

any particular physical organization of the data, and is autonomic in that it does not require any user intervention.

For BHUNT to provide a net benefit, it is crucial that the preceding steps be executed as efficiently as possible. BHUNT will typically be applied to databases comprising many tables with many columns in each table. Because the number of candidate column pairs can grow quadratically with the total number of columns, inexpensive candidate pruning heuristics are key to efficient execution. BHUNT also depends heavily on modern DBMS query sampling and parallel processing technology to deal with the massive amounts of data typically found modern warehouses. Other key elements of the BHUNT scheme include data mining and statistical techniques for identifying the algebraic constraints, and query optimization methods for exploiting the discovered constraints during query processing.

1.4 Related Work

Previous work on automatic methods for learning about data relationships can be categorized according to whether the learning technique is query- or data-driven, and according to the type of information discovered. Query-driven techniques have the nice property that the mined information is, by definition, directly relevant to the user's needs and interests. This narrowed focus often leads to high accuracy. On the other hand, query-driven techniques can result in poor performance during the "warm-up" stage of query processing in which not enough queries have been seen yet. Similar problems arise when the workload starts to change, or when processing a query that is unlike any query previously seen. Indeed, use of query-driven techniques can cause a learning optimizer to "careen towards ignorance" by preferring query plans about which less is known, even if the plans are actually quite inefficient. The reason for this preference is that, in the absence of solid information, an optimizer usually underestimates the cost of a plan, for example, by making unrealistic independence assumptions. Data-driven techniques, though often less precise, complement query-driven techniques and can ameliorate their shortcomings.

One useful type of information about relationships in data is the multidimensional distribution of a set of attributes. A variety of data-driven techniques have been developed for producing "synopses" that capture such distributions in a compressed form; see, for example, [2, 5, 6, 13] and references therein. These methods are based on a scan or sample of the database, which can be initiated by the user or by the system. The methods have somewhat less of an autonomic feel than query-driven methods, because typically the user must specify which attributes to include in each synopsis.

A number of researchers have provided methods for maintaining useful statistics on intermediate query results such as partial joins. The LEO learning optimizer, for example, improves cardinality estimates for intermediate results by observing the data returned by user queries [18]. Techniques proposed by Bruno and Chaudhuri [4] determine the "most important" statistics on intermediate query expressions (SITs) to maintain based on a workload analysis.

The information provided by the foregoing techniques is used by the optimizer to improve the cost estimates of the various access plans under consideration. An alternative set of techniques provides information to the optimizer in the form of rules or constraints. The optimizer can directly use such information to consider alternative access paths. Important types of constraints include *functional dependencies*, *multi-valued dependencies*, *semantic integrity constraints*, and the algebraic constraints considered in the current paper.

Two columns a_1 and a_2 of categorical data obey a functional dependency if the value of a_1 determines the value of a_2. A typical example of a functional dependency occurs when a_1 contains car models and a_2 contains car makes. E.g., a car model value of `Camry` implies a car make value of `Toyota`. A multi-valued dependency is a generalization of a functional dependency that in effect provides a necessary and sufficient condition under which a relation can be decomposed into smaller normalized relations. Mining of functional and multi-valued dependencies is discussed, for example, in [3, 9, 12, 23]. BHUNT's notion of "fuzzy" algebraic constraints is in the spirit of the "approximate" functional dependencies discussed in [9]. Unlike [9], however, BHUNT need only provide probabilistic guarantees on the degree of fuzziness.

Semantic integrity constraints arise in the setting of semantic query optimization. Siegel et al. [17] and Yu and Sun [24], for example, consider query-driven approaches for discovering constraints of the form $A \rightarrow B$ and $JC \rightarrow (A \rightarrow B)$, where JC is a join condition analogous to our pairing rule, and $A \rightarrow B$ is a rule such as `s.city = chicago` $\rightarrow$ `t.weight > 200`. In contrast, we consider algebraic relationships between numerical attributes; an algebraic constraint can be viewed as implying an infinite family of semantic integrity constraints. Moreover, unlike [17, 24], BHUNT's techniques are data-driven, the discovered constraints need not hold for all of the data, and disjunctive constraints are handled naturally and easily.

As indicated previously, the techniques required for generation of candidates in Step 1 of BHUNT are closely related to techniques used in reverse engineering and discovery of entity-relationship (ER) models for legacy databases; see, for example, [3, 12] and references therein. Many of these algorithms rely on information contained in the schema definition — such as primary-key declarations — or in a set of workload queries. Algorithms such as those in [3, 12] execute a sequence of queries involving joins and COUNT(DISTINCT) operations to discover *inclusion dependencies* — an inclusion dependency exists between columns a_1 and a_2 if every value that appears in a_2 also appears in a_1. Our approach incorporates many of these techniques — a key difference lies in our extensive use of sampling and the concomitant decrease in processing-time requirements. BHUNT can use sampling because, in the

context of query optimization, there is no obligation to perfectly capture every possible inclusion dependency.

1.5 Organization of Paper

The remainder of the paper is organized as follows. In Sections 2–4 we describe the steps of the BHUNT scheme in detail, emphasizing applications to query optimization. We then relate (Section 5) our experience with a prototype implementation of BHUNT when run against a large database. In Section 6 we give our conclusions and describe potential extensions of the technology.

2 Generating Candidates

The first step in the BHUNT scheme is to generate candidates of the form $C = (a_1, a_2, P, \oplus)$. Such a candidate corresponds to the set of numbers

$$\Omega_C = \{\, r.a_1 \oplus r.a_2 \colon\ r \in R \,\}$$

when the pairing rule P is a trivial rule $\emptyset_R$ and

$$\Omega_C = \big\{\, r.a_1 \oplus s.a_2 \colon\ r \in R,\ s \in S, \text{ and } (r, s) \text{ satisfies } P \,\big\},$$

when P is a join predicate between tables R and S. We call Ω_C the *induced set* for C. In Examples 1 and 2, it is the points in Ω_C that are histogrammed in Figure 2. We assume that the user has specified a set $O \subseteq \{+, -, \times, /\}$ of allowable algebraic operators.

There is a tension between the desire to be as thorough as possible in identifying candidates and the desire to be as efficient as possible by not examining too many candidates. BHUNT deals with this tension by combining a thorough search strategy with the continual use of pruning heuristics. The precise set of heuristics is flexible and can depend on the goal of the BHUNT analysis. For example, BHUNT can be used for query optimization or for mining; a user would likely employ a more stringent set of heuristics for the former purpose than for the latter.

BHUNT proceeds by first generating a set $\mathcal{P}$ of pairing rules. For each pairing rule $P \in \mathcal{P}$ BHUNT systematically considers possible attribute pairs (a_1, a_2) and operators $\oplus$ with which to construct candidates. At each stage of the process, the pruning heuristics alluded to above are used to keep the number of candidates under control.

2.1 Generating Pairing Rules

BHUNT initializes $\mathcal{P}$ to be the empty set and then adds a trivial pairing rule of the form $\emptyset_R$ for each table R in the database schema.[1] BHUNT then generates nontrivial pairing rules.

The main heuristic underlying the generation of the nontrivial rules is that they should "look like" key-to-foreign-key join predicates, since such joins are the most common type encountered in practice. Specifically, BHUNT first generates a set K of "key-like" columns from among all of the columns in the schema. For each column $a \in K$, BHUNT then tries to identify suitable "foreign-key-like" matching columns from among all of the columns in the schema. That is, BHUNT tries to find all columns related to column a via an inclusion dependency. If n (> 0) such columns $b^{(1)}, b^{(2)}, \ldots, b^{(n)}$ are found, then BHUNT adds the pairing rules $P_1, P_2, \ldots, P_n$ to $\mathcal{P}$, where P_i denotes the predicate "$a = b^{(i)}$" for $1 \le i \le n$.

The columns in K comprise all of the declared primary key columns, all of declared unique key columns, and any column a not of these two types such that

$$\frac{\#\text{rows}(a)}{\#\text{distinctValues}(a)} \le 1 + \epsilon.$$

We call the latter type of column an *undeclared key*. Here ϵ is a pre-specified parameter of BHUNT and the quantities #rows(a) and #distinctValues(a) are obtained from the system catalog. BHUNT additionally requires that the data type of each column in K belong to a user-specified set T of types, where each type in T is suitable for use in equality predicates (e.g., not floating point or BLOB data).

Given a column $a \in K$, BHUNT examines every other column in the schema to find potential matches. A column b is considered a match for column a if the following conditions hold:

1. The data in columns a and b are of the same type.
2. Either
   (a) column a is a declared primary key and column b is a declared foreign key for the primary key, or
   (b) every data value in a sample from column b has a matching value in column a.

The sample used to check the condition in 2(b) need not be large; in our implementation the sample size was set at a few hundred rows.

BHUNT actually can deal with the case in which a declared primary key or declared unique key in K is a compound key of the form $a = (a_1, \ldots, a_m) \in T^m$ for some $m > 1$. In this case, given a compound key $(a_1, \ldots, a_m) \in K$, BHUNT considers as a match every compound attribute $b = (b_1, \ldots, b_m)$ such that columns $b_1, \ldots, b_m$ are in the same table and type(a_i) = type(b_i) for $1 \le i \le m$. Then the conditions in 2(a) and 2(b) are checked to determine whether or not a matches b; of course, "column" now means "compound column," "match" now means "componentwise match," and the pairing rule is a predicate of the form

$$a_1 = b_1^{(i)} \ \texttt{AND} \ \cdots \ \texttt{AND} \ a_m = b_m^{(i)}.$$

To avoid combinatorial explosion of the search space, BHUNT typically does not look for undeclared compound keys.

[1] BHUNT can actually search for algebraic constraints over multiple schemas by simply dealing with the union of the schemas.

As discussed previously, BHUNT applies an adjustable set of pruning rules to limit the number of candidates. The goal of these heuristics is to restrict the set of candidates to those that are likely to generate useful algebraic constraints — a constraint is useful if it can be identified quickly, will arise frequently in practice, and will result in a significant performance improvement. We have found the following set of heuristics for pruning a pairing rule P to be useful in the context of query optimization. (For simplicity, we describe the heuristics when the elements of K are simple, not compound, keys.)

- **Rule 1:** P is of the form $R.a = S.b$ or of the form $\emptyset_R$, and the number of rows in either R or S lies below a specified threshold value. The motivation for this rule is that we only want to look at tables that are important to query performance. Maintaining exception tables over tables that are small initially is probably not a good use of resources. This rule is equivalent to restricting the scope of BHUNT to the M largest tables in the scheme as indicated by system catalog statistics, where M is specified by the user.

- **Rule 2:** P is of the form $R.a = S.b$ with $a \in K$, and the number of distinct values in $S.b$ divided by the number of values in $R.a$ lies below a specified threshold value. In practice, pairing rules that satisfy this condition are likely to be spurious.

- **Rule 3:** P is of the form $R.a = S.b$, and one or both of R and S fails to have an index on any of its columns. This rule is checked when inserting columns into the set K and prior to identifying matches for an element of K. The idea is to preclude columns for which the computational cost of checking the inclusion condition in 2(b) above is high.

- **Rule 4:** P is of the form $R.a = S.b$ with $a \in K$, and $S.b$ is a system-generated key. In this case the pairing rule will be spurious.

2.2 Turning Pairing Rules Into Candidates

For each pairing rule P generated as described above, BHUNT attempts to construct one or more candidates of the form $C = (a_1, a_2, P, \oplus)$. If P is a trivial rule of the form $\emptyset_R$ or is a nontrivial pairing rule that corresponds to a self join of table R, then BHUNT considers every pair of columns in the set $\{\,(a_1, a_2)\colon\ a_1, a_2 \in \mathcal{A}(R) \text{ and } a_1 \neq a_2\,\}$. Here $\mathcal{A}(R)$ denotes the set of columns (i.e., attributes) of R. If P is a nontrivial pairing rule that corresponds to a join of distinct tables R and S, then BHUNT considers every pair $\{\,(a_1, a_2)\colon\ a_1 \in \mathcal{A}(R) \text{ and } a_2 \in \mathcal{A}(S)\,\}$. Each pair (a_1, a_2) is considered in conjunction with the set of possible operators in the user-specified set O. A triple $(a_1, a_2, \oplus)$ is combined with the pairing rule P to form a candidate $C = (a_1, a_2, P, \oplus)$ if the following conditions hold:

1. a_1 and a_2 can be operated on by $\oplus$. E.g., a_1 and a_2 are `float` or `integer` types and $\oplus \in O$, or they are both `date` types and $\oplus \in \{+, -\}$ (since `date` types cannot be multiplied or divided).

2. If the pairing rule P is nontrivial, then a_1 and a_2 cannot correspond to the columns referred to in the pairing rule, since then $r.a_1 = s.a_2$ whenever r and s satisfy P, and any algebraic constraint based on the (a_1, a_2) pairs will be useless.

As when generating pairing rules, additional heuristics can be used to prune the final set of candidates. Examples of useful heuristic pruning rules include the following.

- **Rule 1:** a_1 and a_2 are not of the exact same data type (casting is required).

- **Rule 2:** The fraction of NULL values in either a_1 or a_2 exceeds a specified threshold. The idea is that even if each column has a sufficient number of rows (as in pairing-rule pruning heuristic), the effective number of rows may be small because of NULLs.

- **Rule 3:** Either column a_1 or a_2 is not indexed. The reasoning here is that if there are no indexes, then the database designer probably did not consider columns a_1 and a_2 to be important for query processing performance, so an algebraic constraint based on these columns is not likely to be useful.

3 Identifying Fuzzy Constraints

For each candidate $C = (a_1, a_2, P, \oplus)$ that has been generated using the techniques described in Section 2, BHUNT employs a sampling-based approach to construct a fuzzy algebraic constraint $AC = (a_1, a_2, P, \oplus, I_1, \ldots, I_k)$, where $k \geq 1$. Specifically, BHUNT takes a small sample W_C of the induced set Ω_C and constructs a set of disjoint intervals $I_1, \ldots, I_k$ such that every point in W_C falls within one of the intervals. The sample size is chosen so that with high probability the fraction of points in Ω_C that do not fall within one of the intervals lies below a specified threshold — this small fraction of points corresponds to the set of exception records. We often refer to the I_j's as "bump intervals" because they correspond to bumps in a histogram such as the one in Figure 2. We first describe how bump intervals are constructed from a sample and then describe the sampling process.

3.1 Constructing Bump Intervals

To obtain the bump intervals, BHUNT sorts the n data points in the sampled set W_C in increasing order as $x_1 \leq x_2 \leq \cdots \leq x_n$, and then divides this sequence into disjoint segments. A segmentation $\mathcal{S}$ can be specified as a vector of indices $\big(i(1), i(2), \ldots, i(k)\big)$ that delineate the right endpoints of the segments. That is, the first segment is $x_1, x_2, \ldots, x_{i(1)}$, the second segment is $x_{i(1)+1}, x_{i(1)+2}, \ldots, x_{i(2)}$, and so forth — we take $i(0) =$

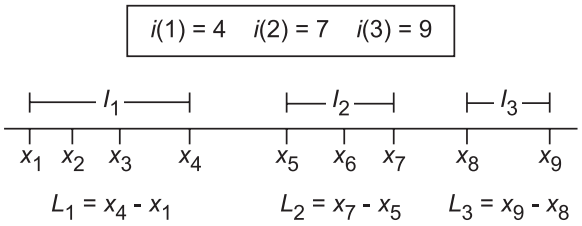

Figure 4: Segmentation of points in W_C

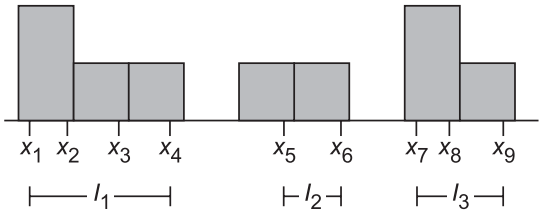

Figure 5: Histogramming method for segmentation

0 and $i(k) = n$. We sometimes call such a segmentation a k-segmentation to emphasize the number of segments. In terms of the foregoing notation, the jth bump interval ($1 \le j \le k$) is given by $I_j = [x_{i(j-1)+1}, x_{i(j)}]$. In other words, the two data points that delineate the segment also delineate the endpoints of the bump interval; see Figure 4. The length of I_j, denoted L_j, is therefore given by $L_j = x_{i(j)} - x_{i(j-1)+1}$. (As discussed below, BHUNT actually adjusts the interval endpoints slightly.)

The optimal-segmentation approach rests on the fact that there is typically a trade-off between the *filtering power* and complexity of an algebraic constraint predicate, where we define filtering power as the sum of the bump interval lengths divided by the range $\Delta = \max_{x\in\Omega_C} x - \min_{x\in\Omega_C} x$ of values for the points in Ω_C. At one extreme, an algebraic constraint comprising many short bump intervals often leads to very selective query predicates that can speed up query processing by cutting down on the number of accesses to the base tables. If the number of intervals becomes too large, however, processing times can start to increase because the many OR clauses in the constraint become expensive to evaluate and, moreover, the query optimization process becomes more complex and hence time consuming. Ideally, BHUNT should choose a segmentation to minimize the overall cost. Unfortunately, it appears hard to quantify the tradeoffs precisely.

As a first cut to this problem, we consider a more ad hoc solution in which we optimize a weighted average of the number of bump intervals and the filtering power of the constraint. That is, for a segmentation $\mathcal{S} = \big(i(1), i(2), \ldots, i(k)\big)$, we set

$$c(\mathcal{S}) = wk + (1-w)\left[\frac{1}{\Delta}\sum_{j=1}^{k} L_j\right], \qquad (5)$$

and find a segmentation $\mathcal{S}$ that minimizes the function c. Here w is a fixed weight between 0 and 1. If w is close to 0 then the optimal segmentation will produce an algebraic constraint with many short intervals; if w is close to 1 then the constraint will comprise a small number of long intervals. The simplest approach to estimating the range Δ is to simply observe the sorted sampled data values $x_1, x_2, \ldots, x_n$ and set $\Delta = x_n - x_1$. The resulting estimate will be low however. A more complicated approach is as follows. Suppose, for example, that we have a candidate $C = (a_1, a_2, P, \oplus)$ in which $\oplus$ is the division operator and all data values are positive. Let a_1^M and a_1^m be the maximum and minimum values in column a_1, and similarly define a_2^M and a_2^m; such parameters (or approximations thereof) can either be obtained from the system catalog or estimated by using the maximum and minimum a_1 and a_2 values in the sample. Then we can estimate Δ as $\Delta \approx (a_1^M/a_2^m) - (a_1^m/a_2^M)$. In any case, once w and Δ are fixed, an optimal segmentation can be easily determined using the following result.

Theorem 1 *Let c be defined as in* (5)*. Then a segmentation that minimizes c is defined by placing adjacent points x_l and x_{l+1} in the same segment if and only if $x_{l+1} - x_l < d^*$, where $d^* = \Delta\big(w/(1-w)\big)$.*

Proof. Denote by $\mathcal{S}^*$ the segmentation described in the theorem, and let $\mathcal{S}$ be an arbitrary segmentation. Observe that $\mathcal{S}$ can be transformed to $\mathcal{S}^*$ through a sequence of steps in which we split and merge segments. Specifically, we examine successive pairs (x_l, x_{l+1}). If $x_{l+1} - x_l < d^*$ but x_l and x_{l+1} are in two different (adjacent) segments, then we place these points in the same segment by merging the two segments. Such a merge decreases the number of bump intervals by 1 and increases the sum of the interval lengths by $x_{l+1} - x_l$. Thus the change in the cost function is

$$\begin{aligned} c_{\text{new}} - c_{\text{old}} &= -w + (1-w)\frac{x_{l+1} - x_l}{\Delta} \\ &< -w + (1-w)\frac{d^*}{\Delta} = 0. \end{aligned}$$

Similarly, if $x_{l+1} - x_l \ge d^*$ but x_l and x_{l+1} are in the same segment, then we place these points in different segments by splitting the original segment. In this case the change in the cost function is

$$\begin{aligned} c_{\text{new}} - c_{\text{old}} &= w - (1-w)\frac{x_{l+1} - x_l}{\Delta} \\ &\le w - (1-w)\frac{d^*}{\Delta} = 0. \end{aligned}$$

Since the cost is nonincreasing at each step, it follows that $c(\mathcal{S}^*) \le c(\mathcal{S})$. Since $\mathcal{S}$ is arbitrary, the desired result follows. □

When dealing with discrete data types such as DAY or INTEGER, BHUNT actually uses the value $\max(d^*, 1+\epsilon)$ for segmentation, where ϵ is a small positive constant.

An alternative approach to segmenting the values in W_C is to identify "natural" clusters of the points, using any of

the many well known clustering techniques available; see Section 14.3 in [8]. In this context, the "gap statistic" of Tibshirani [19] can be used to choose the number of segments. The drawback of such an approach is the high computational cost involved — since BHUNT generates many candidate algebraic constraints, it is important to keep the cost of computing each constraint very low.

One inexpensive natural clustering method that has worked well in experiments is based on a histogramming approach. The idea is for BHUNT to construct a histogram using an appropriate bucket width. Adjacent nonempty buckets are then merged to form an initial set of bump intervals, and then each of these intervals is trimmed if necessary so that the interval endpoints each coincide with one of the x_i's; see Figure 5. We use $2h(n)$ buckets, where $h(n) = (2n)^{1/3}$ is the "oversmoothing" lower bound as described in Section 3.3 of [16]. Use of this number of buckets approximately minimizes the "asymptotic mean integrated squared error" of the histogram when the histogram is viewed as an estimator of the underlying density function of the data. Other methods, such as those in [7], can be used to determine the number of buckets, but at a significantly higher cost. If the histogramming method creates a segment consisting of a single point, then BHUNT adds to the algebraic constraint a bump interval centered around the data point and having a width corresponding to the oversmoothing rule.[2]

In general, BHUNT can specify an upper limit on the number of bumps allowed. If this limit is exceeded, then BHUNT greedily merges the closest bump intervals, then the closest bump intervals of those remaining, and so forth.

For real-valued data, it is beneficial to expand the interval widths by a few percent (merging any bumps that overlap after the expansion). To see the reason for this, suppose that we have taken a sample and consider the right endpoint of the rightmost bump interval. This point corresponds to the maximum value seen in the sample W_C. Ideally, the right endpoint should correspond to the maximum value in Ω_C. Typically, the observed maximum value grows as the logarithm of the sample size, so a good deal of additional sampling is required to increase the right endpoint to the correct value. Directly expanding the endpoint slightly achieves the same effect with much less effort. Similar reasoning applies to the other bump-interval endpoints.

Note that for each bump interval we can use the fraction of sample points in W_C that lie within the interval as an estimate of the fraction of all points in Ω_C that lie within the interval. These "selectivities" can be used by the optimizer for purposes of cost estimation. Standard techniques can be used to estimate the precision of the selectivities.

[2]We do not simply ignore such a data point because, with high probability, this sample point "represents" many points in Ω_C. Moreover, the penalty for accidentally basing a constraint on an outlier point is small, at least in the context of query optimization.

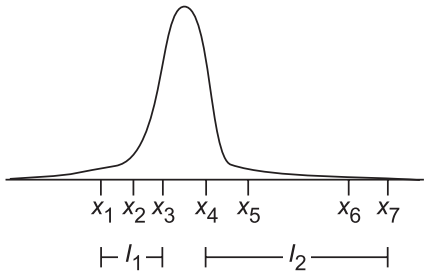

Figure 6: A low quality segmentation

3.2 Choosing the Sample Size

As mentioned previously, BHUNT computes algebraic constraints based on small samples of the data. For a candidate $C = (a_1, a_2, P, \oplus)$, the specific type of sampling depends on the form of the pairing rule P. If P is a trivial rule of the form $\emptyset_R$, BHUNT samples the data by obtaining randomly-selected rows from R. If P is a join predicate between a key-like column a_1 in R and a foreign-key-like column a_2 in S, then BHUNT samples by obtaining randomly selected rows from S — for each sampled row of S, BHUNT then obtains the matching row of R as determined by P.

BHUNT tries to choose the sample size so as to control the number of exceptions, and hence the size of the exception tables. Unfortunately, the segmentation methods that BHUNT uses are so complicated that the distribution of the number of exceptions is extremely hard to compute. BHUNT's approach is to compute the target sample size based on the behavior of a "randomized" approximation to the actual segmentation algorithm. This randomized algorithm takes as input parameters a target number of bump intervals k and a sample size n. The randomized algorithm takes a simple random sample of n points from Ω_C with replacement, and then chooses a k-segmentation randomly and uniformly from among all possible k-segmentations. The idea is that the target sample size for the actual algorithm should be comparable to the ideal sample size for the randomized algorithm. In fact, the latter sample size should be a rough upper bound for the former sample size, because the randomized algorithm is likely to yield somewhat less effective bump intervals. This loss of effectiveness arises because the randomized algorithm will sometimes choose a low quality segmentation such as the one in Figure 6; for the displayed segmentation, the region around the mode of the true distribution (displayed above the horizontal axis) is not covered by a bump interval.

The distribution of the number of exceptions for the randomized algorithm is given by Theorem 2 below. Recall that the beta distribution with parameters α and β is defined by

$$\text{Beta}(t; \alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \int_0^t u^{\alpha-1}(1-u)^{\beta-1}\, du,$$

for $t \geq 0$, where Γ is the standard gamma function given

by $\Gamma(t) = \int_0^\infty x^{t-1}e^{-x}\,dx$.

Theorem 2 *Let F be the random fraction of elements of Ω_C that lie outside of the set of bump intervals $I = I_1 \cup \cdots \cup I_k$ produced by the randomized algorithm from a sample of n data points. Then*

$$P\{\, F > x \,\} \le \mathrm{Beta}(1-x; n-k, k+1). \qquad (6)$$

Proof. The randomized algorithm is statistically equivalent to an algorithm in which we first randomly choose a segmentation and then sample the data points; we analyze this latter version of the algorithm. First consider the conditional distribution of F, given the k-segmentation. Let $X_1, X_2, \ldots, X_n$ be the simple random sample of size n drawn with replacement from Ω_C and sorted so that $X_1 \le X_2 \le \cdots \le X_n$. Also let $\mathcal{S} = \big(i(1), i(2), \ldots, i(k)\big)$ be a fixed segmentation of $X_1, \ldots, X_n$ so the bump intervals are defined by $I_j = [X_{i(j-1)+1}, X_{i(j)}]$ for $1 \le j \le k$. If $X_1, X_2, \ldots, X_n$ were independent and identically distributed (iid) random variables with a common continuous distribution function, then a result due to Tukey [21] would imply that $P\{\, F > x \mid \mathcal{S} \,\} = \mathrm{Beta}(1-x; n-k, k+1)$. By an argument essentially as in [15], we can drop the continuity assumption provided that we replace "$=$" with "$\le$" in the foregoing equality. The intuitive reasoning behind this latter argument is that discrete data accumulates at a relatively small set of locations, so that the probability that a data point falls outside a specified collection of intervals is lower than in the case of continuous data. Since the case of a discontinuous distribution function corresponds precisely to simple random sampling with replacement from a finite population, we have shown that

$$P\{\, F > x \mid \mathcal{S} \,\} \le \mathrm{Beta}(1-x; n-k, k+1)$$

for the randomized algorithm. Now observe that the right side of the above expression does not depend on the specific form of the k-segmentation. The desired conclusion now follows by unconditioning on $\mathcal{S}$. □

Suppose that we wish to use the randomized algorithm to construct an algebraic constraint having k bump intervals, and we want to be assured that, with probability at least p, the fraction of points in Ω_C that lie outside the bump intervals is at most f. It follows from Theorem 2 that the constraint should be based on at least n^* samples, where n^* solves the equation

$$\mathrm{Beta}(1-f; n-k, k+1) = 1-p.$$

In the following, we denote this solution by $n^* = n^*(k)$ to emphasize the dependence on the number of bump intervals. We can determine $n^*(k)$ somewhat painfully by solving the above equation numerically. Alternatively, Scheffé and Tukey [14] have developed an approximation to the inverse of the beta distribution which leads to the following approximation for $n^*(k)$:

$$n^*(k) \approx \frac{\chi^2_{1-p}(2-f)}{4f} + \frac{k}{2}. \qquad (7)$$

Here χ^2_α is the $100\alpha\%$ percentage point of the χ^2 distribution with $2(k+1)$ degrees of freedom — this quantity can be quickly and easily computed using, e.g., formulas 26.4.17 and 26.2.23 in [1]. Scheffé and Tukey assert that the error in the approximation is at most 0.1%; our own experiments indicated that the maximum error is at most 0.2%, but this degree of accuracy is more than sufficient for our purposes.

For the actual segmentation algorithm, we use $n^*(k)$ as our target sample size for creating an algebraic constraint with k bumps. Of course, we do not know *a priori* the value of k. The fact that $n^*(k)$ is increasing in k, however, suggests the following iterative sample size procedure, given prespecified values of f and p:

1. (Initialization) Set $i = 1$ and $k = 1$.
2. Select a sample size $n = n^*(k)$ as in (7).
3. Obtain the sample and compute an algebraic constraint. Observe the number k' of bump intervals.
4. If $n \ge n^*(k')$ or $i = i_{\max}$, then exit; else set $k = k'$ and $i = i + 1$, and go to step 2.

The quantity $i_{\max}$ is a parameter of the algorithm. In our experiments, the sample size always converged within two or three iterations. The actual algorithm used by BHUNT is slightly more complicated in that it takes NULLs into account: we maintain an estimate of the fraction q of NULL values of $a_1 \oplus a_2$ and scale up the sample size by a factor of $1/q$.

In many commercial database systems, rows are sampled using a Bernoulli sampling scheme. For row-level Bernoulli sampling at rate p, each row is included in the sample with probability p and excluded with probability $1-p$, independently of the other rows. When there are a total of N rows in the table, the resulting sample size is random but equal to Np on average; the standard deviation of the sample size is $\big(Np(1-p)\big)^{1/2}$. Page-level Bernoulli sampling is similar, except that entire pages of rows are included or excluded. For the low sampling rates typical of BHUNT applications, the Bernoulli sampling schemes behave almost identically to simple random sampling with replacement, so that the foregoing development still applies. In this connection, we note that at first glance there may be cause for concern about the applicability of Theorem 2 when page-level Bernoulli sampling is employed and the data in column a_2 is "clustered" on disk, so that there is a strong relationship between the value in column a_2 and the page on which the corresponding row is located. In practice, however, the resulting values of $a_1 \oplus a_2$ in Ω_C are rarely clustered, so that clustering does not pose a real problem to our methodology.

Our implementation of BHUNT uses a conservative procedure to guard against samples that are too small due to Bernoulli fluctuations in the sample size. The idea is to boost the Bernoulli sampling rate so that, under the boosted rate, the target sample size lies three standard deviations

```
CREATE TABLE exceptions(
   CHAR(3) o-oid, CHAR(3) d-oid,
   DATE o-sdate, DATE d-ddate, TIME d-dtime)

INSERT INTO exceptions AS
(SELECT orders.orderID, deliveries.orderID,
   orders.shipDate, deliveries.deliveryDate,
   deliveries.deliveryTime
FROM orders, deliveries
WHERE orders.orderID = deliveries.orderID
AND NOT (
   (deliveryDate BETWEEN shipDate + 2 DAYS
      AND shipDate + 5 DAYS)
   OR (deliveryDate BETWEEN shipDate + 12 DAYS
      AND shipDate + 19 DAYS)
   OR (deliveryDate BETWEEN shipDate + 31 DAYS
      AND shipDate + 35 DAYS))
)
```

Figure 7: Creating the exception table for Example 1

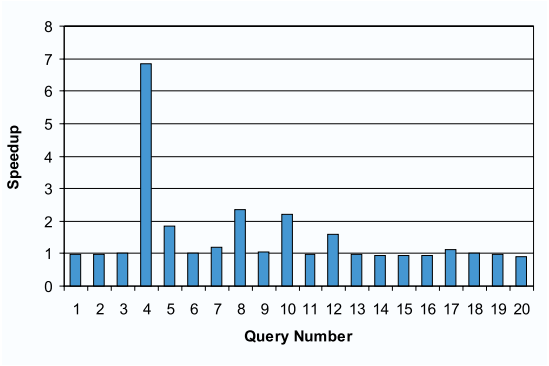

Figure 8: Experimental results

below the expected sample size. Thus the probability of seeing a sample size below the target size is very small. If p is the target sampling rate, then the boosted rate is given by $q \approx p + 3(p/N)^{1/2}$, where N is the number of either rows or pages, depending on whether row-level or page-level Bernoulli sampling is used, respectively.

For the various reasons outlined above, the sample size procedure tends to be conservative, especially for data with many duplicate values, such as integers or dates. In preliminary experiments, the mean fraction of exceptions was less than or equal to the user-specified fraction in virtually all cases. In the case of discrete data, we were able to reduce the target size by a factor of 5 and still keep the number of exceptions at or below the target value f.

4 Exploiting the Constraints

As discussed previously, the algebraic constraints found by BHUNT can be used in multiple ways, such as for data mining and for improving query processing performance. In the latter context, for example, the constraints can be passed to a system-configuration tool, so that the DBA receives guidance on how to reconfigure the data, or the system can perform the reconfiguration automatically. We focus here on the direct use of discovered constraints by the query optimizer.

In query optimization mode, BHUNT automatically partitions the data into "normal" data and "exception" data. In general, this can be done in a variety of ways, for example by physically partitioning the data or by using partial indexes. In our initial implementation, BHUNT creates exception tables.

The `WHERE` clause in an SQL statement for creating the exception table contains the predicate (if present) in the pairing rule P, as well as the logical negation of the algebraic constraint predicate. For example, the exception table for the constraint in Example 1 might be specified as shown in Figure 7. To reduce the costs incurred during optimization and query processing, it may be desirable to maintain a single exception table for all constraints that involve a specified pairing rule P.

Because of resource limitations, it may be necessary to retain only the "most important" constraints when constructing the exception tables. One way to rank the algebraic constraints — especially appropriate when $\oplus$ is the subtraction operator — is to arrange them in decreasing order of (estimated) filtering power as defined previously.

During query processing, each query is modified, if possible, to incorporate the discovered constraints. The modified query is run against the original data, the original query is run against the data in the exception table, and the two sets of results are combined. The implementation details for the optimization process are rather involved, and a detailed description is beyond the current scope of the paper. We simply note here that the algorithm builds on standard query processing technology.

5 Empirical Results

In this section we describe our experience running a prototype implementation of BHUNT against a large database. The database exceeds 2.3Tb in size and a schema similar to the TPC-D schema described in [20]. The largest table had in excess of 13.8 billion rows while the next biggest table had in excess of 3.45 billion rows.

For the test database, which contains 7 years of (synthetic) retail data, the most notable constraints that BHUNT discovered are:

```
lineitems.shipDate BETWEEN orders.orderDate
   AND orders.orderDate + 4 MONTHS

lineitems.received BETWEEN lineitems.shipDate
   AND lineitems.shipDate + 1 MONTH
```

Other constraints are implied by the two above, and none of the discovered constraints were fuzzy. The time to discover the algebraic constraints was approximately 4 minutes. Figure 8 shows the performance impact of BHUNT on 20 different queries. For each query, the figure shows the ratio of the elapsed processing time without BHUNT to the elapsed time with BHUNT.

As can be seen, there is a performance improvement for half of the queries, with significant improvements for 25% of the queries. There were no significant performance decreases for any of the queries. The most dramatic speedup — by a factor of 6.83 — occurred for Query 4. For this lat-

ter query, the number of accesses to the large `lineitem` table were reduced by a factor of about 100.

6 Conclusions and Future Work

We have presented BHUNT, a new data-driven mining technique for discovering fuzzy hidden relationships among the data in a RDBMS. BHUNT provides the discovered relationships in the form of constraint predicates that can be directly used by a query optimizer. In this context, the BHUNT technique can be used to automatically create data structures and modify queries to obtain speedups. Preliminary experiments on a large database show that BHUNT can potentially provide significant performance improvements when processing massive amounts of data; further experimentation is currently in progress.

In future work, we plan to improve the basic algorithm in a number of ways. To improve scalability and performance, we plan to incorporate techniques as in [3] to exploit the transitivity of the inclusion relationship and reduce the number of sampling queries when searching for pairing rules. We also plan to combine techniques similar to those in [3] with more elaborate heuristics to reduce the likelihood of generating spurious pairing rules caused by the presence of system-generated keys.[3] Finally, we plan to consider operators other than the simple algebraic ones.

We are also investigating various extensions of the basic technology. One potential extension is to discover (fuzzy) functional dependencies. Recall our example: car model determines car make. To apply BHUNT to this problem, we can hash the values of a_1 and a_2 into a very large set of numbers. Then we look at the values of $a_1 - a_2$. In our example, the histogram of x values will have a spike at $\#(\texttt{Toyota}) - \#(\texttt{Camry})$ but not at $\#(\texttt{Toyota}) - \#(\texttt{Explorer})$. If the number of spikes is much less than the number of distinct values in a_1 times the number of distinct values in a_2, then a functional dependency is indicated. The locations of the spikes provide information about the specific nature of the dependency.

Another extension is to handle data organizations other than relational. For example, it may be possible to discover useful schema and data relationships in XML repositories using our approach.

Acknowledgements

Hamid Pirahesh originally suggested the algebraic constraint problem. Haider Rizvi and his colleagues Qi Cheng, Shu Lin, Wen-Bin Ma, Richard Sidle, Ashutosh Singh, Jason Sun, and Calisto Zuzarte played key roles both in modifying the query optimizer to exploit the discovered constraints and in running the experiment described in Section 5.

[3] Although the majority of such spurious rules are pruned by Rule 4 in Section 2.1, problems can still arise when an element of the set K defined in Section 2.1 is a system-generated key.

References

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions*. Dover, New York, 1972. Ninth printing.

[2] D. Barbarà, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. Ioannidis, H. V. Jagadish, T. Johnson, R. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey data reduction report. *IEEE Data Engrg. Bull.*, 20:3–45, 1997.

[3] S. Bell and P. Brockhausen. Discovery of constraints and data dependencies in databases. In *Proc. Europ. Conf. Machine Learning (ECML-95)*, Lecture Notes in Artificial Intellegence 914, pages 267–270, Berlin, 1995. Springer-Verlag.

[4] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proc. 2002 ACM SIGMOD Intl. Conf. Managment of Data*, pages 263–274. ACM Press, 2002.

[5] A. Deshpande, M. N. Garofalakis, and R. R. Independence is good: Dependency-based histogram synopses for high-dimensional data. In *Proc. 2001 ACM SIGMOD Intl. Conf. Managment of Data*, pages 199–210. ACM Press, 2001.

[6] M. N. Garofalakis and P. B. Gibbons. Wavelet synopses with error guarantees. In *Proc. 2002 ACM SIGMOD Intl. Conf. Managment of Data*, pages 476–487. ACM Press, 2002.

[7] P. Hall and E. J. Hannan. On stochastic complexity and nonparametric density estimation. *Biometrika*, 75:705–714, 1988.

[8] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag, New York, 2001.

[9] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Tiovonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42:100–111, 1999.

[10] IBM Research. Autonomic computing, 2003. http://www.research.ibm.com/autonomic.

[11] Microsoft Research. The AutoAdmin project, 2003. http://research.microsoft.com/dmx/autoadmin.

[12] J.-M. Petit, F. Toumani, J.-F. Boulicaut, and J. Kouloumdjian. Towards the reverse engineering of denormalized relational databases. In *Proc. 12th Intl. Conf. Data Engrg.*, pages 218–227. IEEE Computer Society Press, 1996.

[13] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proc. 23rd Intl. Conf. Very Large Data Bases*, pages 486–495, 1997.

[14] H. Scheffé and J. W. Tukey. A formula for sample sizes for population tolerance limits. *Ann. Math. Statist.*, 15:217, 1944.

[15] H. Scheffé and J. W. Tukey. Nonparametric estimation. I. Validation of order statistics. *Ann. Math. Statist.*, 16:187–192, 1945.

[16] D. W. Scott. *Multivariate Density Estimation: Theory, Practice, and Visualization*. Wiley, New York, 1992.

[17] M. Siegel, E. Sciore, and S. Salveter. A method for automatic rule derivation to support semantic query optimization. *ACM Trans. Database Syst.*, 17:563–600, 1992.

[18] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO — DB2's LEarning Optimizer. In *Proc. 27th Intl. Conf. Very Large Data Bases*, pages 19–28, 2001.

[19] R. Tibshirani. Estimating the number of clusters in a data set via the gap statistic. *J. Roy. Statist Soc. B*, 63:411–423, 2001.

[20] Transaction Processing Performance Council (TPC). *TPC Benchmark D (Decision Support) Standard Specification, Revision 2.1*. San Jose, CA, 1998. http://www.tpc.org/tpcd.

[21] J. W. Tukey. Nonparametric estimation II. Statistically equivalent blocks and tolerance regions—The continuous case. *Ann. Math. Statist.*, 18:529–539, 1947.

[22] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *Proc. 28th Intl. Conf. Very Large Data Bases*, pages 20–34, 2002.

[23] S. K. M. Wong, C. J. Butz, and Y. Xiang. Automated database schema design using mined data dependencies. *J. Amer. Soc. Inform. Sci.*, 49:455–470, 1998.

[24] C. T. Yu and W. Sun. Automatic knowledge acquisition and maintenance for semantic query optimization. *IEEE Trans. Knowledge Data Engrg.*, 1:362–375, 1989.